Using Annotated Suffix Trees for Fuzzy Full Text Search

Abstract. A method for fuzzy full text search is proposed. The method follows a popular two-stage scheme with a novel second stage: a preliminary search stage using an n-gram inverted index and, at the second stage, relevance checking between the query and documents using frequency annotated suffix trees (ASTs). The ASTs are built for all documents of the collection off-line. The method is compared with two popular fuzzy text retrieval techniques, one using n-gram inverted indexing with Levenshtein distance checking and signature hashing, and the other being Lemur, a popular toolkit for language modelling and information retrieval. For computational experiments we use "Reuters 21578" text collection and a collection of USPTO patents. Our AST-based method generally leads to accuracy scores that are similar to those obtained by the winner, the Levenshtein distance-based method. However, our method significantly outperforms the Levenshtein distance based method over speed. Therefore, when using both criteria, the accuracy and speed, simultaneously, the AST-based method has shown significant advantages.

Keywords: text retrieval; full text search in databases; fuzzy text search; annotated suffix tree (AST); n-gram indexing

1 Introduction

The issue of raising efficiency of fuzzy text search is one of the most important in information retrieval and database theory. There are very few embedded database solutions or light-weight software tools to perform such type of search. This often necessitates using databases for document storage only and looking for third-party search engines to extend functionality.

There are many text retrieval methods, including boolean search [1], topic modelling approaches [2], inverted index-based algorithm. The last one is the most popular approach: it applies the idea of feature-to-document mapping and provides an efficient way to retrieve all the documents corresponding to a given feature [1]. There are many classical information retrieval systems, such as Lemur toolkit [26]. But using "feature-based" methods, i.e. based on words or combinations of words, may cause some troubles too. For example, detecting different forms of the same word can be a rather complicated problem. Therefore usually such methods require large dictionaries or special stemming or lemmatization algorithms [3, 5]. This makes them less applicable in some cases. We cannot make light-weight cross-language software solutions, for instance, in the case of development databases or database extensions.

In real situations text features may contain mistakes or other inaccuracies. This fact aggravates the issue. In such a case we are in the realm of fuzzy text search. There are several methods of fuzzy text search; an excellent survey can be found in [16], it describes and compares several classical methods and their further modifications. The simplest one of these methods uses so-called extended inverted index at which some inaccurate forms of features are appended to the index. Other methods are based on pronunciation rules of the language, as for example, Soundex algorithms [6]. Also there is the n-gram index approach: some substrings (of length n) are included in index instead of features). This approach is implemented in some databases as a tool for simple fuzzy full text search [15].

The more advanced methods apply distance between features calculation: Levenshtein (sometimes called "edit") distance [9, 10], Damerau-Levenshtein distance, Jaro-Winkler distance, or BK-trees [8] and further modifications. There are combined variants of distance-based methods, for instance, together with index approaches. For example, in [10] and group of similar papers were proposed several methods based on the idea of preliminary substituting (or extending) features from search query by features from a text dictionary using n-gram index [12]. These techniques provide very high search accuracy levels, but their time complexity is often quite high.

One more popular method is the so-called signature hashing: it applies special hashing procedure for features [13, 16]. The main idea here is the following: both the inaccurate feature and the original one are to be mapped into the same hash. Only hashes of features are stored into the index. The main disadvantage of this method is possible collisions: some absolutely unrelated features can be mapped into the same hash value.

Also there are word-embedding models, for example, Word2Vec [27] by Google, based on neural networks. Word2Vec often shows excellent results in synonyms and mistakes detection. However, these methods require training, that is, additional computations and additional memory. Furthermore, if a text collection under consideration was changed (for example, collection extending may imply new mistakes in words), these methods require some kind of retraining. Apparently, these fact makes Word2Vec and similar approaches inapplicable in some situations.

In this paper we propose a method of language-independent fuzzy search, which does not require training. It can be easy implemented, for example, as a tool for fuzzy full text search in databases. This method is based on a different approach to text representation, which was developed in a series of B. Mirkin and E. Chernyak's papers (for example, [19]). This approach uses the so-called annotated suffix trees (AST) as a tool for document processing. The authors used this approach to obtain the so-called string-to-document relevance score. We describe our method and compare it with some other popular techniques, specifically, the classical Levenshtein distance method together with n-gram inverted indexing [10], and the signature hashing method from [13, 14].

It should be pointed out that all the methods under consideration can be applied to any data represented by texts.

2 Method

We constructed a method for text search based on annotated suffix trees (AST). AST can be used for representing individual texts. A full description of a method for constructing AST can be found in papers [19] and (with some improvements) [21], here we give a brief description.

AST is a weighted rooted tree, which is used for storing text fragments and their frequencies. One of tree nodes is the root of the tree, an empty tree contains the root only. Other nodes contain text symbols and corresponding frequencies that are called annotations. To construct an AST for a text string we extract all suffixes from this string. A k-suffix of a string $x = x_1x_2 \dots x_N$ of length N is a continuous substring $x_k = x_{N-k+1}x_{N-k+2} \dots x_N$. For example, 3-suffix of string $INFORMATION$ is a substring ION, and 5-suffix, $ATION$. Note that N-suffix of any string is a whole string. Each AST node is assigned a symbol and the so-called annotation (frequency of the substring corresponding to the path from the root to the node including the symbol at the node). The root node of AST has no symbol or annotation.

An algorithm for constructing AST for any given string $x = x_1x_2 \dots x_N$ is described below.

1. Initialize an AST consisting of a single node, the root: T.
2. Find all the suffixes of the given string:
   $\{x^k = x_{N-k+1}x_{N-k+2} \dots x_N | k = 1, 2, \dots, N\}$.
3. For each suffix x^k find its maximal overlap, that is, coinciding path from the root, in T: $x^{k_{max}}$. At each node of the path for $x^{k_{max}}$ increase the annotation by 1. If the length of the overlap $x^{k_{max}}$ is less than k, we extend the path by adding new nodes corresponding to symbols from the remaining part of this suffix. Annotations of all the new nodes are equal to 1.

An example of AST for string $ABCBA$ is shown in Figure 1.

Fig. 1. An example of AST

Having an AST T built, we can score the string-to-document relevance by using it. To do this we follow [19] by computing the conditional probability of node u in T:

$$p(u) = \frac{f(u)}{\sum\limits_{v \in T: parent(u) = parent(v)} f(v)}, \tag{1}$$

where $f(u)$ is the frequency annotation of the node u. Using the formula above we can calculate the probability of node u relative to all its siblings. For each suffix x_k of the string x the relevance score $s(x_k, T)$ is defined as:

$$s(x_k, T) = \frac{1}{k_{max}} \sum_{i=1}^{k_{max}} p(x_i^k). \tag{2}$$

Finally, the relevance score of the string is expressed by the following formula:

$$S(x, T) = \frac{1}{N} \sum_{k=1}^{N} s(x_k, T). \tag{3}$$

For example, the relevance score of substring BAC and AST constructed for string $ABCBA$ is 0.35. Thus, AST approach allows to obtain string-to-document relevance scores. We split any document into strings (usually we use 2-3 consecutive words as one string), create an empty AST for the document and add these strings in AST sequentially. We carry out this procedure for each document of the given collection, build AST for each document, after that we have all AST's constructed (1 AST for 1 document). If we want to search for relevant documents for the given query, we have to compute string-to-document relevance scores for all documents and choose the most relevant ones.

To improve the performance of this algorithm we use a special fragment (also called n-gram) inverted index. The index has the following form:

$$\begin{cases} f_1 \leftarrow [n_{11}, \dots, n_{1m_1}] \\ f_2 \leftarrow [n_{21}, \dots, n_{2m_2}] \\ \dots \\ f_K \leftarrow [n_{K1}, \dots, n_{Km_K}], \end{cases} \tag{4}$$

where features f_i are document substrings (n-grams, 3–4 sequential symbols), and n_{ij} are numbers of corresponding documents which contain these features. K is a number of features. This approach allows to decrease the number of calculations by taking into account only "probably good documents" (for further relevance evaluating using AST). The final search algorithm consists of the following steps:

1. Split a query into substrings and select corresponding documents from inverted index.
2. Calculate string-to-document relevance score for these documents using AST
3. Sort the documents by relevance score.

Note, if we denote m as a length of a search query (i.e. count of symbols), the time complexity of calculating string-to-document relevance score is $O(m^2)$. It is independent from document length. In contrast, the time complexity of Levenshtein distance procedure (for a query consisting of 1 feature) is $O(N \cdot m \cdot n)$, where N is a number of document features, n is average feature length. This complexity can be reduced by considering no more than k differences – $O(N \cdot k \cdot min(m, n))$. Nevertheless, the last expression depends still on document length. Therefore, the direct applying Levenshtein distance approach to fuzzy full text search can be inappropriate in a case of large documents. Also note both these methods require some additional memory.

3 Experiments and results

We carry out computational experiments using real data. We used laptop with 2.0 GHz dual-core processor and 4 GB RAM, running under Ubuntu 15.04 operating system for our aims.

In our comparison we have considered AST-based method, Levenshtein distance (LD) based method (both with n-gram inverted indexing) signature hashing (SH) algorithm and retrieval method from Lemur toolkit. To test these methods we take 2 collections of text documents: "Reuters-21578" collection [23] (#1) and collection of USPTO patents [24] (#2), both in English. The first one (containing about 20000 articles) was used for quality measurement and performance (speed of search procedure) testing. The second collection was obtained from a set of patents in pdf format. We downloaded several thousand pdf documents from official site [24] and chose 8000 largest documents from this set. This collection was used for performance testing only.

All search queries were obtained from collection's features and their sequences using substitutions by variants containing mistakes. We used Wikipedia lists of common English misspellings for these[25]. This gives real user mistakes and eliminates the need in surrogate cases. On the other hand, when we randomly "injured" some features in documents, the procedure gives exactly the same results. Examples are: "red aple" (the correct one is "red apple"), "gramatical rules" ("grammatical rules"), "abritrary positive number" ("arbitrary positive number"). Overall, 120 queries (per each collection) was prepared. All decisions about document relevance were combined from the fact whether the document contains the correct query and expert judgements. Using Collection #1 we have calculated precision and recall for the results obtained for all the queries using standard formulas (which are shown below).

$$Recall = \frac{|Relevant\ documents\ provided\ by\ the\ system|}{|All\ relevant\ documents|} \tag{5}$$

$$Precision = \frac{|Relevant\ documents\ provided\ by\ the\ system|}{|All\ documents\ provided\ by\ the\ system|} \tag{6}$$

Average precision of the methods is shown in Table 1. Here, 5-document and 10-document levels mean the number of documents, which are used to calculate the precision, an important quality metric, because in many problems only most relevant documents are needed.

Table 1. 5 and 10-document level precision

Document level	AST-based method	LD-based method	SH-based method	Lemur method
5	0.81	**0.82**	0.69	0.54
10	0.64	**0.66**	0.53	0.46
Average	0.73	**0.74**	0.61	0.50

To show a level of search quality and see an interplay between the precision and recall, let us use so-called 11-pt precision-recall curves [7]. This curve shows a precision level for a given recall level at the same search method. Usually one uses 11 levels of recall (from 0.0 to 1.0 with a step equal to 0.1). Precision-recall curves for different search methods are shown by Figure 2 below.

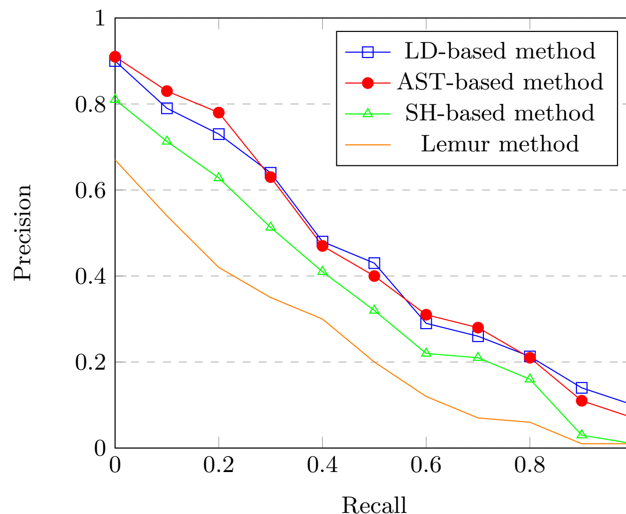

Fig. 2. Precision-recall curves for the methods under consideration

So, AST-based method slightly concedes Levenshtein distance-based but it is significantly better than signature hashing method and Lemur method. Note the last one is not specially designed for fuzzy search, but nevertheless it shows visible results. To test the performance of the search methods under consideration we used both collections, #1 and #2. The AST-based retrieval method appears to be slower than Lemur and those based on signature hashing, but it is significantly faster than the Levenshtein distance-based method. The results are presented in Table 2.

Table 2. Average time of query processing for different methods (CPU time, sec.)

Collection	AST-based method	LD-based method	SH-based method	Lemur method
#1	0.0210	0.1320	0.0110	**0.0080**
#2	0.0130	0.0870	0.0060	**0.0050**
Average	0.0170	0.1100	0.0090	**0.0065**

Also we tried to use AST with different string lengths used at the tree construction phase (usually we use 2-3 words as one string), in a range from 1 word to 4-5 words. On the basis of using the Collection #2 for experiments we can conclude: "stretching out" of strings increases the query performing time (see Table 3).

We can improve the performance of the algorithm by imposing restrictions on the maximum admissible length of AST strings. However, in such a case we

Table 3. Influence string length used in AST and query processing time (CPU time, sec.)

AST with 1 word per string	AST with 2-3 words per string	AST with 4-5 words per string
0.010	0.013	0.022

can lose helpful conjunctions between words and impair the quality metrics of the algorithms. We are going to investigate this in further work. Also one can note the time of given query processing for the AST-based method depends on the length of this query. This can be explained by the need to apply the string-to-document relevance score formula.

4 Conclusions

The main purpose of this project is to show that the annotated suffix tree (AST) techniques can be competitively applied to fuzzy text retrieval problems. We have developed an AST-based search method and compared it with two popular fuzzy text retrieval techniques. Our experiments demonstrate that the AST-based method provides both competitive quality and good performance, in contrast to the other methods under consideration. Our future work should include experimentation over the admissible length of the strings entering the AST-based processing. Also this method should be easy to implement as a light-weight database plug-in.

Acknowledgments. I wish to thank Prof. B. Mirkin for his advice and valuable comments.

References

1. Manning, C. D., Raghavan, P., Schutze, H. *Introduction to information retrieval.* Cambridge university press, Cambridge (2008).
2. Wei, X., & Croft, W. B. (2006, August). *LDA-based document models for ad-hoc retrieval.* In Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval (pp. 178–185). ACM.
3. Lovins, J. B. (1968). *Development of a stemming algorithm.* MIT Information Processing Group, Electronic Systems Laboratory.
4. Porter, M. F. (1980). *An algorithm for suffix stripping.* Program, 14(3), pp. 130–137.
5. Willett, P. (2006). *The Porter stemming algorithm: then and now*. Program, 40(3), pp. 219–223.
6. Holmes, D., & McCabe, M. C. (2002, April). *Improving precision and recall for soundex retrieval.* In Information Technology: Coding and Computing, 2002. Proceedings. International Conference on (pp. 22–26). IEEE.

7. Ageev M., Kuralenok I., Nekrest'janov I. (2004) *Oficial'nye metriki ROMIP'2004* [ROMIP'2004 Official Metrics]. Proceedings of the ROMIP 2004, pp. 142–150.
8. Burkhard, W. A., & Keller, R. M. (1973). *Some approaches to best-match file searching*. Communications of the ACM, 16(4), pp. 230–236.
9. Yujian, L., & Bo, L. (2007). *A normalized Levenshtein distance metric*. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 29(6), pp. 1091–1095.
10. Zobel, J., & Dart, P. (1995). *Finding approximate matches in large lexicons*. Softw., Pract. Exper., 25(3), pp. 331–345.
11. D'elena, D. F., & Martinez, A. E. (1998). *Method and system for performing non-boolean search queries in a graphical user interface*. U.S. Patent No. 5,842,203. Washington, DC: U.S. Patent and Trademark Office.
12. Rangarajan, V., & Ravichandran, N. (1998). *System and method for portable document indexing using n-gram word decomposition*. U.S. Patent No. 5,706,365. Washington, DC: U.S. Patent and Trademark Office.
13. Chegrane, I., & Belazzougui, D. (2014). *Simple, compact and robust approximate string dictionary*. Journal of Discrete Algorithms, 28, pp. 49–60.
14. Boytsov L. (2001). *Ispol'zovanie heshirovanija po signature dlja poiska po shodstvu*. [Using signature hashing for approximate matching]. Prikladnaja matematika i informatika, VMiK MGU, No 8, 2001, pp. 135–154.
15. Korotkov, A. (2010). *Database index for approximate string matching*. In Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering (No. 4). FGBUN ISP RAN.
16. Boytsov, L. (2011). *Indexing methods for approximate dictionary searching: Comparative analysis*. Journal of Experimental Algorithmics (JEA), 16, pp. 1–11.
17. Mirkin B. (2011). *Metody klaster-analiza dlya podderzhki prinyatiya reshenii: obzor*. [Cluster Analysis for Decision Making: Review]. Working paper WP7/2011/03, Moscow: HSE.
18. Chernjak, E. L., Mirkin, B. G. (2013). *Ispol'zovanie resursov Interneta dlja postroenija taksonomii*. [Using the Internet for Taxonomy Constructing]. Proceedings of the AIST 2013. Nacional'nyj otkrytyj universitet INTUIT, pp. 36–48.
19. Mirkin B. G., Chernjak E. L., Chugunova O. N. (2012). *Metod annotirovannogo suffiksnogo dereva dlja ocenki stepeni vhozhdenija strok v tekstovye dokumenty*. [Annotated Suffix Tree as a Way of String-To-Document Score Evaluating]. Business Informatics, no. 3 (21), pp. 31–41.
20. Malkov Y. et al. (2014). *Approximate nearest neighbor algorithm based on navigable small world graphs*. Information Systems, No 45, pp. 61–68.
21. Dubov M., Chernjak E. (2013). *Annotirovannye suffiksnye derev'ja: osobennosti realizacii*. [Annotated Suffix Trees: Implementation Features]. Proceedings of the AIST 2013. NOU INTUIT, pp. 49–57.
22. Frolov D.S. (2015) Annotated suffix tree as a way of text representation for information retrieval in text collections. Business Informatics, no. 4 (34), pp. 63–70.
23. Reuters-21578 Test Collection. Available at *http://www.daviddlewis.com/resources/testcollections/reuters21578/* (accessed 12 Jan 2015).
24. USPTO patent full-text databases. Available at *http://patft.uspto.gov/* (accessed 14 Jan 2015).
25. Wikipedia Lists of common misspellings (English). Available at *https://en.wikipedia.org/wiki/Wikipedia: Lists_of_common_misspellings* (accessed 15 July 2015).
26. Lemur Project. Available at *http://www.lemurproject.org/* (accessed 31 Jan 2015).
27. Word2Vec - Google Code. Available at *https:// code.google.com/p/word2vec/* (accessed 20 Jan 2015).